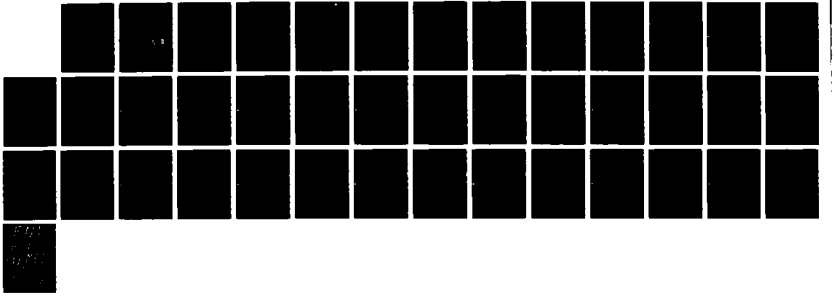
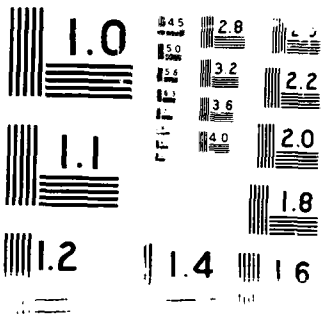
AD-A190 146 PATTERN-DIRECTED INVOCATION WITH CHANGING EQUALITIES (U) MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL INTELLIGENCE LAB Y A FELDMAN ET AL. MAY 88 AI-M-1017 1/1

UNCLASSIFIED N00014-85-K-0124 F/G 12/9 NL




1.0
1.1
1.2
1.4
1.6
2.8
3.2
3.6
4.0
2.2
2.0
1.8

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered

(4)

REPORT DOCUMENTATION PAGE

AD-A198 146

1. REPORT NUMBER: AIM-1017

2. GOV

NS FORM

BER

DTIC FILE COPY

4. TITLE (and Subtitle)

Pattern-Directed Invocation with Changing Equalities

5. TYPE OF REPORT & PERIOD COVERED

memorandum

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)

Yishai A. Feldman and Charles Rich

8. CONTRACT OR GRANT NUMBER(s)

N00014-85-K-0124

9. PERFORMING ORGANIZATION NAME AND ADDRESS

Artificial Intelligence Laboratory
545 Technology Square
Cambridge, MA 02139

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS

Advanced Research Projects Agency
1400 Wilson Blvd.
Arlington, VA 22209

12. REPORT DATE

13. NUMBER OF PAGES

14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)

Office of Naval Research
Information Systems
Arlington, VA 22217

15. SECURITY CLASS. (of this report)

UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution is unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

DTIC ELECTE AUG 29 1988 S D E

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

automated reasoning
truth maintenance
demons

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The interaction of pattern-directed invocation with equality in an automated reasoning system gives rise to a completeness problem. In such systems, a demon needs to be invoked not only when its pattern exactly matches a term in the reasoning data base, but also when it is possible to create a variant that matches. An incremental algorithm has been developed, which solves this problem without generating all possible variants of terms in the data base. The algorithm is shown

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered

to be complete for a class of demons, called *transparent* demons, in which there is a well-behaved logical relationship between the pattern and the body of the demon. Completeness is maintained when new demons, new terms, and new equalities are added in any order. Equalities can also be retracted via a truth maintenance system. The algorithm has been implemented as part of a reasoning system called BREAD.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Memo No. 1017 May 1988

Pattern-Directed Invocation With Changing Equalities

by

Yishai A. Feldman and Charles Rich

Abstract

The interaction of pattern-directed invocation with equality in an automated reasoning system gives rise to a completeness problem. In such systems, a demon needs to be invoked not only when its pattern exactly matches a term in the reasoning data base, but also when it is possible to create a variant that matches. An incremental algorithm has been developed, which solves this problem without generating all possible variants of terms in the data base. The algorithm is shown to be complete for a class of demons, called *transparent* demons, in which there is a well-behaved logical relationship between the pattern and the body of the demon. Completeness is maintained when new demons, new terms, and new equalities are added in any order. Equalities can also be retracted via a truth maintenance system. The algorithm has been implemented as part of a reasoning system called BREAD.

Submitted to the Journal of Automated Reasoning

The research described here was conducted at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research has been provided in part by the National Science Foundation under grant IRI-8616644, in part by the IBM Corporation, in part by the NYNEX Corporation, in part by the Siemens Corporation, and in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-85-K-0124. Yishai Feldman was supported in part by a grant from the Bantrell Charitable Trust.

The views and conclusions contained in this document are those of the authors, and should not be interpreted as representing the policies, neither expressed nor implied, of the National Science Foundation, of the IBM Corporation, of the NYNEX Corporation, of the Siemens Corporation, or of the Department of Defense.

Contents

Acknowledgements

We would like to thank Walter Hamscher, David McAllester, Reid Simmons, Dilip Soni, and Richard Waters for their suggestions and careful reading of drafts of this paper. Any remaining errors are ours.

1 Introduction

Pattern-directed invocation and equality reasoning are two techniques commonly used in automated reasoning systems. This paper describes a problem that arises when these two techniques are combined, and gives a solution.

1.1 A Simple Example of the Problem

Suppose we are reasoning about a function, f, which obeys the axiom

$$\forall x \; f(0, x) \geq 0.$$

A typical implementation of this axiom is a demon with the pattern $f(0, ?x)$ and a body that asserts $f(0, ?x) \geq 0$.[1] This demon can be thought of as instantiating the universally quantified axiom above for appropriate terms in the reasoning data base. For example, when the term $f(0, c)$ is added to the data base, this demon is invoked and asserts $f(0, c) \geq 0$.

In the presence of equality, however, the proper conditions for invoking a demon become more complicated. Suppose, for example, that the term $f(a, b)$ is added to the reasoning data base and the equality $a = 0$ holds. In the usual algorithm for pattern-directed invocation, nothing will happen, since there is no term in the data base that exactly matches the pattern of the demon. This is undesirable, however, because we would like the reasoning system in this situation to deduce $f(a, b) \geq 0$, which follows logically from the axiom above. We describe this problem as a lack of *completeness* in the pattern-directed invocation.

Note, however, that if the term $f(0, b)$ is somehow created in this situation, the desired deduction will be made. The term $f(0, b)$ exactly matches the pattern of the demon, which then asserts $f(0, b) \geq 0$. Since $f(0, b)$ is equal to $f(a, b)$ by substitution of equals (0 for a), equality reasoning further deduces that $f(a, b) \geq 0$.

Thus, a brute-force approach to the lack of completeness in standard pattern-directed invocation is to close the data base under substitution of equals. This approach is not feasible, however, because the numbers of terms generated grows exponentially with the number of equalities, and is infinite in the case of recursively defined equalities.

[1] Variables are denoted, as usual, by the prefix "?".

The algorithm developed in this paper solves the completeness problem by generating a subset of all possible substitutions, based on an analysis of the patterns of existing demons. Furthermore, the algorithm is incremental. New demons, new terms, and new equalities can be added in any order, and completeness will be maintained. Equalities can also be retracted.

1.2 Outline of the Paper

Section 2 provides additional background of the completeness problem and sets the stage for the rest of the paper, including defining the terminology that will be used. In addition to pattern-directed invocation and equality reasoning, Section 2 introduces a third component, truth maintenance, into the environment in which the algorithm operates. Among other things, this means that completeness needs to be maintained when equalities are retracted.

Section 3 formally defines completeness for a given data base of terms and set of demons. Examples are given of demons with simple and more complex patterns to illustrate the subtlety of the issues in the most general case. Finally, Section 3 defines the notion of a transparent demon and then proves a key theorem that is the basis of the algorithm of Section 4.

Section 4 develops an algorithm that incrementally maintains completeness given new demons, new terms, and changing equalities. The development starts with a simple generalization of the standard algorithm for pattern-directed invocation, and then adds special processing for demons with complex patterns. Section 4 also discusses termination of the algorithm and some extensions that improve its performance.

Section 5 concludes with a discussion of alternate approaches and related work.

2 Background and Environment

The completeness problem described briefly above first came to our attention when we began to use McAllester's Reasoning Utility Package (RUP) [5] in our research related to reasoning about programs [8]. Among other facilities, RUP included equality reasoning, a primitive pattern-directed invocation mechanism (demons were associated with operator symbols), and a truth maintenance system. We used these facilities to build up a library of demons

for reasoning about various algebraic properties of operators. Since our reasoning also involved asserting and retracting equalities between operators, we immediately began to run into the incompleteness problem.

In our successor to RUP, called BREAD (for Basic REAsoning Device), we have extended RUP to support a full pattern-matching language with variables, and have implemented the complete algorithm for pattern-directed invocation described in this paper. BREAD also includes further evolved versions of RUP's truth maintenance and equality reasoning components. BREAD is now the kernel of a general-purpose knowledge representation and reasoning system, called FRAPPE [2], which is itself the foundation of a special-purpose system for reasoning about programs, called CAKE [7].

Pattern-directed invocation, equality reasoning, and truth maintenance are common components of many reasoning systems. In order to facilitate the incorporation of our algorithm into other systems, we attempt in this paper, as much as possible, to abstract away from details that are idiosyncratic to BREAD. In this vein, the following sections summarize the essential properties of pattern-directed invocation, equality reasoning, and truth maintenance that are relevant to our algorithm.

2.1 Pattern-Directed Invocation

Pattern-directed invocation is a common technique used in reasoning systems, wherein a given procedure is to be applied to every term of a particular form in the data base. The procedure is associated with a pattern. The combination is typically called a *demon*; the procedure is called the *body* of the demon.

Whenever a new term is added to the data base, it is matched against the pattern of each demon. If the pattern matches, the body of the demon is applied to the matching term. (It is often convenient to also supply the set of bindings to the pattern variables as an argument to the body, although this can be computed from the matching term). Similarly, whenever a new demon is added to the system, its pattern is matched against all terms in the data base, and the body of the demon is applied to each matching term. In most implementations of pattern-directed invocation, elaborate indexing structures are used to make these matching processes efficient.

In general, both terms and patterns can be hierarchical. An *atomic* term is a single symbol, such as f or 0. Non-atomic terms are built up by applying operators to arguments, recursively, such as $f(0, b)$ or $P(f(a, b), c)$.

Patterns are written using the same notation as terms, except that variables may appear in some positions. (Technically, a pattern may have no variables. This degenerate case occurs in several examples in Section 4.4.) Patterns are not, however, stored in the term data base—they are only associated with demons.

A *flat* pattern is a list of terms and variables, such as $f(0, ?x)$ or $P(f(a, b), ?x)$. A flat pattern has all of its variables at "top level". A pattern with variables at other than top level is called a *nested* pattern. For example, $P(f(?x, ?y), ?y)$ is a nested pattern. The pattern $f(?x, ?y)$ is called a *subpattern* of $P(f(?x, ?y), ?y)$.

The algorithms in this paper treat all the positions of patterns and terms symmetrically. Therefore, there is nothing to prevent the use of patterns with variables in the operator position, such as $?f(0, b)$.

The notation used for demons in this paper is illustrated below:

$$\langle\ f(0, ?x)\ ,\ \lambda t\,.\,\text{assert}(t \geq 0)\ \rangle.$$

This is the demon introduced in the simple example problem in Section 1.1. The body of the demon, which may be an arbitrary procedure, is applied to a matching term t. Assert is the primitive that makes a premise.

The demon above is an example of a common form of restricted demon, in which the body is simply a call to assert, with its argument constructed from parts of the input term. This form of demon is usually defined using a notation that makes the assert implicit and uses variables to define the body, as illustrated below:

$$\textbf{Rule}\ f(0, ?x) \Longrightarrow f(0, ?x) \geq 0.$$

With this form of demon, the bindings returned by the matching process are substituted in the right-hand side of the rule before it is asserted. In implementations of pattern-directed invocation, such as BREAD, which allow arbitrary procedures in demon bodies, this rule notation can be compiled into a procedure.

A standard algorithm for pattern-directed invocation is given in Figure 1. It will serve as the starting point for the complete algorithm developed in Section 4. Note that in Figure 1 and elsewhere, we use the notation $a \equiv b$ to mean "a is identical to b" (i.e., they are spelled the same), versus $a = b$, which means "a equals b" in the current equality relation.

```
1  function match(p, t, B)
       [[ Match pattern p to term t with bindings B. ]]
2      case
3          p is a variable:
4              if p is bound to s in B
5                  then if s ≡ t
6                           then return B      [[ same value ]]
7                           else fail          [[ different value ]]
8                  else return B ∪ {p ← t}     [[ new binding ]]
9          p is a pattern p0(p1, ..., pn):
10             if t is a term t0(t1, ..., tn)
11                 then for i in {0, ..., n} do B ← match(pi, ti, B)
12                      return B               [[ all positions match ]]
13                 else fail
14         p is a constant:
15             if p ≡ t then return B
16                  else fail

17     function try(t, p, d)
           [[ Match term t against pattern p and apply body d if successful. ]]
18         B ← match(p, t, ∅)
19         if match succeeded
20             then apply d to t [and B]

21     function add-term(t)
           [[ Add new term t to data base. ]]
22         for each demon ⟨ p , d ⟩
23             try(t, p, d)
24         add t to data base

25     function add-demon(p, d)
           [[ Add demon with pattern p and body d. ]]
26         for each term t in data base
27             try(t, p, d)
28         add ⟨ p , d ⟩ to demons
```

Figure 1. A standard pattern-directed invocation algorithm. Note that the match procedure returns a set of bindings when it succeeds. Failure to match at any level is assumed to propagate to the top level.

2.2 Equality Reasoning

Reasoning with changing equalities involves two main tasks. The first task is to maintain the equivalence classes of the equality relation under transitivity and reflexivity. These classes need to be joined or split when equalities are asserted or retracted.

The second task is to guarantee that all the variants of a term are in the same equality class. A *variant* of the (non-atomic) term t is any term derived from t by substitution of equals. Variants of t are therefore equal to t. For example, $f(g(a,b),c)$ is a variant of $f(d,e)$, given the equalities $d=g(a,b)$ and $e=c$. The task of keeping variants in the same equality class is sometimes called the *congruence closure* problem, because it has been shown to be reducible to the problem of computing the congruence closure of a relation on a graph (for a formal definition, see [6]).

The set of all possible variants of a term grows exponentially with the number of applicable equalities. For example, given $a=b$ and $b=c$, there are eight possible variants of $f(a,b)$. Moreover, the set of all possible variants is potentially infinite. For example, there are infinitely many variants of $h(a)$, given the recursive equality $a=h(a)$.[2] It is therefore obviously not feasible to solve the completeness problem by creating all possible variants of all terms in the data base.

Note that equality reasoning does not, by itself, add new terms to the reasoning data base. All that equality reasoning does is update the equality classes on existing terms when equalities change, and place newly created terms into the correct equality class.[3]

The equality reasoning algorithm used in BREAD is due to McAllester [4].

2.3 Truth Maintenance

Nonmonotonic reasoning is a common feature of many kinds problem solving, such backtracking search or hypothetical reasoning. In such cases, propositions in the reasoning data base sometimes need to be retracted along with

[2]Such equalities are also often called "circular", because they correspond to cycles in the graph of the equality relation.

[3]The equality reasoning in BREAD actually does create new terms for efficiency reasons related to retraction of equalities. However, these are specially managed so that they do not interact with pattern-directed invocation.

their consequences. The collection of facilities to support this kind of reasoning has come to be called a *truth maintenance system* (TMS) [1, 3].[4]

The most basic function of a TMS is to keep track of which propositions in the reasoning data base are supported by the current set of premises. A *proposition* is a boolean-valued term. For example, both $a+1$ and $a \geq 0$ are terms, but only $a \geq 0$ is a proposition. A *premise* is a proposition that is believed without support.

Reasoning procedures interface with a TMS by asserting and retracting premises and installing *dependencies* between propositions. To *assert* a proposition means to make it a premise. This is a monotonic change—it can only expand the set of supported propositions. To *retract* a premise means to remove it from the current set of premises. This is a nonmonotonic change—the set of supported propositions can shrink as a result.

The main task of the TMS is to incrementally keep track of which propositions in the reasoning data base are supported, as premises are asserted and retracted. A proposition is supported if and only if all of the propositions it depends on are either premises, or are supported. Like equality reasoning, a TMS does not, by itself, create or destroy terms.

There are many other important issues in the implementation and use of a TMS not discussed here, because they are not relevant to the algorithm developed here.

A reasoning system that is constructed using a TMS must obey a stringent discipline of installing the appropriate dependencies whenever deductions are made. For example, the following section describes the interface between equality reasoning and truth maintenance.

2.4 Equality and Truth Maintenance

In this paper, we assume (as is the case in BREAD) that equalities are propositions in the reasoning data base. This does not contradict the claim that equality reasoning *per se* creates no new terms. For example, if some reasoning process is interested in whether a is equal to c, given the premises $a=b$ and $b=c$, it would typically add the proposition $a=c$ to the data base.

[4] This is somewhat a misnomer, since the system isn't really concerned so much with what is true, but rather the reasons for believing things. Some attempts have been made to relabel these systems as "belief maintenance", or "reason maintenance", but none of these terms has stuck.

Equality reasoning then installs the appropriate dependencies (transitivity), from which the TMS concludes that $a = c$ is supported.

A similar process takes place with reasoning by substitution of equals. In the example of Section 2.2, the equality between variants

$$f(g(a,b),c) = f(d,e)$$

depends on the equalities used in the substitution:

$$\begin{aligned} d &= g(a,b), \\ e &= c. \end{aligned}$$

Note, however, that even if these supporting equalities are retracted, it is still possible for $f(g(a,b),c) = f(d,e)$ to be a premise or be supported for other reasons, i.e., the two terms could be equal but not variants. This will turn out to be an important consideration in the development of algorithm for pattern-directed invocation.

Another part of the interface between between equality reasoning and the TMS concerns the special case of equality between propositions (boolean-valued terms): A proposition is supported if it is equal to a supported proposition. For example, the proposition Q can depend on the proposition R and the equality $Q = R$. Similarly, the proposition $P(a)$ can depend on the proposition $P(b)$ and the equality $P(a) = P(b)$, which itself depends on the equality $a = b$.

3 Completeness and Transparency

We begin this section with a formal definition of completeness. The notion of equivalent variants is then introduced and used to define a transparent demon as one that has logically equivalent results when applied to equivalent variants. The section culminates in a theorem establishing an implementable constraint on pattern-directed invocation that guarantees completeness for transparent demons. The algorithm developed in Section 4 maintains this condition incrementally.

3.1 Completness

The intuitive notion of completeness is that we want to create enough variants of terms in the data base to make use of all the knowledge in the demons. We can define this more formally as follows.

> **Definition.** A data base of terms is *complete* with respect to a given set of demons if and only if applying demons to matching variants of terms in the data base does not change the equality theory of the data base.

The *equality theory* of a data base is the set of propositions that follow logically from supported propositions in the data base by the axioms of equality (transitivity, symmetry, reflexivity, and substitution of equals). Note that this defines completeness for a given (fixed) data base; the purpose of the algorithm in Section 4 is to guarantee completeness incrementally when demons and terms are added and equalities change.

3.2 Equivalent Variants

The definition of completeness above can be satisified trivially (at least in the absence of recursively defined equalities) by applying demons to all possible variants of every term in the data base. In this section we define an equivalence relation on variants that helps characterize which variants are necessary for completeness, and which are not.

Let us return to the simple example demon introduced in Section 1.1:

$$\textbf{Rule } f(0,?x) \Longrightarrow f(0,?x) \geq 0.$$

Suppose the term $f(a,b)$ is in the data base, and $a=0$ and $b=1$. Given this equality relation, the set of possible variants of the term $f(a,b)$ is

$$\{f(a,1), f(0,b), f(0,1)\}.$$

The first variant, $f(a,1)$, doesn't match the pattern of the demon, so it clearly does not need to be created. The second and third variants both match the pattern. Only one of these variants is necessary, however, to conclude that

$$f(a,b) \geq 0.$$

If the demon is applied to the variant $f(0,b)$, it will assert

$$f(0,b) \geq 0,$$

from which $f(a,b) \geq 0$ follows by equality. Alternatively, if the demon is applied to $f(0,1)$, it will assert

$$f(0,1) \geq 0.$$

from which $f(a,b) \geq 0$ follows by equality. (The algorithm in Section 4 chooses $f(0,b)$ for reasons which will be explained later.)

In the case of flat patterns, only one variant is necessary to achieve completeness. The main complexity in the completeness problem arises in the case of nested patterns. To illustrate, consider the following demon

$$\textbf{Rule } P(h(?x)) \implies P(h(?x)) \rightarrow Q(?x),$$

which implements the axiom[5]

$$\forall x\ P(h(x)) \rightarrow Q(x).$$

Suppose the term $P(c)$ is in the data base, and

$$\begin{aligned} c &= h(a), \\ c &= h(b), \end{aligned}$$

but it is not the case that $a{=}b$ (i.e., $h(a)$ and $h(b)$ are equal, but not variants). The set of matching variants of $P(c)$ in this situation is

$$\{P(h(a)),\ P(h(b))\}.$$

In this case, *both* of these variants are needed for completeness. If only the variant $P(h(a))$ is created, then the demon will assert $P(h(a)) \rightarrow Q(a)$. If $P(c)$ is true, $Q(a)$ will be deduced, but $Q(b)$ will not. Similarly, if only the variant $P(h(b))$ is created, then the demon will assert $P(h(b)) \rightarrow Q(b)$, but $Q(a)$ will not be deduced.

To make things even more complex, now suppose in addition that

$$\begin{aligned} a &= d, \\ b &= e. \end{aligned}$$

There are now two more matching variants possible: $P(h(d))$ and $P(h(e))$. However, if the demon has already been applied to the $P(h(a))$ and $P(h(b))$, there is no need to create these two new variants, since applying the demon to them would result in the assertions $P(h(d)) \rightarrow Q(d)$ and $P(h(e)) \rightarrow Q(e)$, which follow by substitution from the assertions above.

The reason why $P(h(a))$ and $P(h(b))$ are both needed is that they are equal by substitution at intermediate levels in the pattern $P(h(?x))$. In

[5] A concrete instance of this schema is the axiom $\forall x\ \log x \geq 0 \rightarrow x \geq 1$.

contrast, $P(h(a))$ and $P(h(d))$ are equal by substitution in positions occupied by variables in the pattern. Thus if the demon body does not depend on the internal structure of the variable bindings it gets, it cannot distinguish between $P(h(a))$ and $P(h(d))$ (this notion is formalized in the next section).

In general, we can define the following equivalence relation on the set of variants that match a given pattern. Note that this definition involves only terms and patterns. The pattern of a demon is the only declarative information about a demon that will be available to the pattern-directed invocation algorithm.

> **Definition.** Two variants matching a given pattern are *equivalent* with respect to the given pattern and an equality relation if and only if they differ only by substitution of equals at positions occupied by variables in the pattern.

Thus in the first example above, $f(0,b)$ and $f(0,1)$ are equivalent variants with respect to the pattern $f(0,?x)$ and the given equality relation.

In the case of flat patterns, all matching variants are equivalent. This follows because two terms that match the same flat pattern can differ only at positions occupied by variables in the pattern.

In the second example, the set of matching variants of $P(c)$ is partitioned into equivalence classes as follows:

$$\Big\{ \{P(h(a)), P(h(d))\},\ \{P(h(b)), P(h(e))\} \Big\}.$$

To guarantee completeness with respect to this demon, we need to apply the demon to one variant from each of these classes.

In the next section, the notion of equivalent variants is used together with the notion of transparent demons to prove the key theorem of the paper.

3.3 Transparent Demons

The two example demons in the preceding sections both have the property that equivalent variants are redundant. It is possible, however, to write demons that do not have this property. For example, consider the following demon as an alternate implemention of the axiom introduced in Section 1.1.

$$\langle\ f(?y,?x)\ ,\ \lambda t\,.\,\textbf{if}\ t_1 \equiv 0\ \textbf{then}\ \text{assert}(t \geq 0)\ \rangle.$$

(The symbols t_0, t_1, and t_2 denote the operator, first, and second arguments of the term t, respectively.) The problem with this demon is that it is hiding some of its pattern matching inside of the body, where it is inaccessible to the pattern-directed invocation algorithm. For example, given $a=0$ and $b=1$, the following set of matching variants are all equivalent:

$$\{f(a,b), f(a,1), f(0,b), f(0,1)\}.$$

However, if the demon is applied only to $f(a,b)$ or $f(a,1)$, the desired conclusion $f(a,b) \geq 0$ will not be deduced, because in both cases the condition in the body will be false and the demon will make no assertion.

The property that this demon lacks, and which the other example demons in the paper thus far possess, can defined formally as follows:

> **Definition.** A demon is *transparent* if and only if it results in data bases with the same equality theory, when applied to equivalent variants with respect to its pattern and a given data base.

One might think to define a transparent demon more simply as one that results in the *same* data base, when applied to equivalent variants. For example, the the demon above is not transparent, because the data base resulting when it is applied to the matching term $f(0,b)$ includes the proposition $f(0,b) \geq 0$, whereas the data base resulting when it is applied to the equivalent matching variant $f(a,b)$ does not.

However, consider the example of a transparent demon in Section 3.2. The data base resulting when it is applied to the matching term $f(0,b)$ is not the same as the data base resulting when it is applied to the equivalent matching variant $f(0,1)$. (One data base includes the proposition $f(0,b) \geq 0$, while the other includes the proposition $f(0,1) \geq 0$, and not vice versa.) However, the equality theory of the two data bases is the same (given $a=0$ and $b=1$).

Any demon written in the rule notation introduced in Section 2.1 is guaranteed to be transparent. In general, the behavior of the body of a transparent demon must not be conditional on the values of its bindings, i.e., the subterms of the input term that correspond to variable positions in the pattern. (BREAD does not attempt to automatically analyze demon bodies to check this property.)

The ultimate motivation for the definition of transparency above is the following theorem, which establishes an implementable constraint on pattern-directed invocation that guarantees completeness.

Theorem 1 (Completeness) *A data base of terms is complete with respect to a given set of transparent demons if each demon is applied to (at least) one member of each equivalence class of matching variants of every term in the data base.*

The proof of this theorem is as follows. Consider applying a demon d to a matching variant v of a term t in the data base. The antecedent of the theorem states that d has already been applied to an equivalent matching variant v'. Since d is transparent, the data base resulting from applying d to v has the same equality theory as the data base resulting from applying d to v'. Therefore the data base is complete.

We assume in the rest of this paper that demons are transparent, unless stated otherwise.

3.4 Opaque Demons

An *opaque* (i.e., non-transparent) demon can often be made transparent by removing conditionals from the body and changing the pattern. For example, consider the following opaque demon, which implements the idempotent law, $\forall x\ f(x,x)=x$, for the operator f:

$$\langle\ f(?x,?y)\ ,\ \lambda t\,.\,\textbf{if}\ t_1 \equiv t_2\ \textbf{then}\ \text{assert}(t=t_1)\ \rangle.$$

This demon can be rewritten in the following transparent form:

$$\textbf{Rule}\ f(?x,?x)\implies f(?x,?x)=?x.$$

It might occur to the reader to try to make the opaque demon above transparent by replacing $t_1 \equiv t_2$ by $t_1 = t_2$ in the body. Although, technically speaking, the demon in this form would be transparent, it would not be a very good implementation of the idempotency law, for two reasons. First, the law would only be instantiated for applications of f that are added to the data base *after* the equality between t_1 and t_2 is established. Second, if the equality between t_1 and t_2 is later retracted, the proper dependencies have not been installed to cause $t=t_1$ to be retracted also.

If there is more than one conditional in the body of a demon, it may be necessary to break it up into several transparent demons, corresponding to the different cases. For example, the following opaque demon

$$\begin{array}{l}\langle\ f(?x,?y)\ ,\ \lambda t\,.\,\textbf{case} \\ \qquad\qquad\qquad\qquad t_1 \equiv 0\text{: } case_0 \\ \qquad\qquad\qquad\qquad t_1 \equiv 1\text{: } case_1\ \rangle\end{array}$$

can be rewritten as the following two transparent demons:

$$\begin{array}{l}\langle\ f(0,?y)\ ,\ \lambda t\,.\,case_0\ \rangle \\ \langle\ f(1,?y)\ ,\ \lambda t\,.\,case_1\ \rangle\end{array}$$

Not all opaque demons can be handled in this way. For example, consider the following opaque demon:[6]

$$\langle\ f(?x,?y)\ ,\ \lambda t\,.\,\textbf{if}\ \mathrm{lt}(t_1,t_2)\ \textbf{then}\ \mathrm{assert}(t \geq 0)\ \rangle.$$

There is no way to make this demon transparent simply by giving it a different pattern or expanding it into a (finite) number of demons with the same body. To guarantee complete use of the knowledge in this demon, it would have to rewritten as

$$\textbf{Rule}\ f(?x,?y) \implies ?x < ?y \rightarrow f(?x,?y) \geq 0.$$

This version of the demon has the disadvantage that it will instantiate the right-hand side for *all* applications of f. However, it does enable proving that $f(a,b) \geq 0$ when only the relationship between a and b is known, not their specific numerical values.

4 Development of the Algorithm

In this section we develop a complete algorithm for pattern-directed invocation in two stages. The first stage develops a complete algorithm for the case of flat patterns. The second stage reduces the case of nested patterns to flat patterns by introducing intermediate demons.

Within the development of the flat pattern case, there are three steps. The first step, in Section 4.1, introduces a simple generalization of the standard pattern-directed invocation algorithm, and shows that it maintains completeness when new demons or terms are added. In Section 4.2, an additional

[6] Note that "lt" is a test in the programming language, which is executed when the body of the demon is executed. This should not be confused with "<", which is the operator of a term in the data base.

procedure is introduced to maintain completeness when new equalities are asserted. Finally, in Section 4.3, we prove that no action is required when equalities are retracted, due to the use of the closest matching variant.

Section 4.4 introduces the procedure the flattens nested patterns and proves that completeness is maintained in this case. Section 4.5 discusses termination of the algorithm. Section 4.6 describes some improvements to the algorithm that reduce the number of redundant proof paths.

4.1 A Simple Generalization

Figure 2 shows a simple generalization of the standard algorithm for pattern-directed invocation. The most basic change is that the identity ($\equiv$) tests in lines 5 and 15 of the standard matching procedure have been replaced by equality ($=$) tests. Thus, when match-w-equality succeeds, it does not necessarily mean that the given term matches the given pattern, but only that the given term or *some variant* matches the given pattern. The bindings returned by the matcher indicate a matching term.

In line 21, the try-w-equality procedure, which is called whenever a demon or new term is added, uses the bindings returned by the matcher to create a matching variant, if necessary. (Substitute(p, B) returns the pattern resulting from substituting the bindings B in the pattern p.) For example, when $f(a, b)$ is matched against the pattern $f(0, ?x)$, with a=0, match-w-equality succeeds and returns the bindings $\{?x \leftarrow b\}$. Try-w-equality then creates the matching variant $f(0, b)$.[7]

Note that when a new matching variant term is created, the add-term-w-equality procedure is *not* called. This feature of the algorithm is crucial to its termination (see Section 4.5). The new term is in a sort of "limbo" state, in which it exists from the standpoint of the equality system, but not from the standpoint of pattern-directed invocation. This term will be added to the data base only if add-term-w-equality is explicitly called with it as an argument.

Note also (line 22) that, when the matching variant (v) is already in the data base, try-w-equality does *not* apply the given demon, unless $v \equiv t$. This is because under these conditions, the given demon will have been applied

[7]One might also think of simply applying the demon directly to the given term when the match succeeds, and never bothering to create the matching variant. This approach is discussed in Section 5.1.

```
1  function match-w-equality(p, t, B)
       [[ Match pattern p to term t with bindings B with equalities. ]]
2      case
3          p is a variable:
4              if p is bound to s in B
5                  then if s = t
6                           then return B        [[ same value ]]
7                           else fail            [[ different value ]]
8                  else return B ∪ {p ← t}       [[ new binding ]]
9          p is a pattern p_0(p_1, ..., p_n):
10             if t is a term t_0(t_1, ..., t_n)
11                 then for i in {0, ..., n} do
12                          B ← match-w-equality(p_i, t_i, B)
13                      return B                 [[ all positions match ]]
14                 else fail
15         p is a constant:
16             if p = t then return B
17                     else fail

18     function try-w-equality(t, p, d)
           [[ Match term t against pattern p with equalities
             and apply body d to t or matching variant, if successful. ]]
19         B ← match-w-equality(p, t, ∅)
20         if match succeeded
21             then v ← the-term(substitute(p, B))
22                  if v is not in data base or v ≡ t
23                      then apply d to v [and B]
```

Figure 2. A simple generalization of the standard pattern-directed invocation algorithm in Figure 1, which is complete for flat patterns only. (Continued on next page.)

```
24 function add-term-w-equality(t)
       [[ Add new term t to data base with equalities. ]]
25     for each demon ⟨ p , d ⟩
26         try-w-equality(t, p, d)
27     add t to data base

28 function add-demon-w-equality(p, d)
       [[ Add demon with pattern p and body d with equalities. ]]
29     for each term t in data base
30         try-w-equality(t, p, d)
31     add ⟨ p , d ⟩ to demons
```

Figure 2. (Continued)

to v when try-w-equality was called on v, t, and d. It is necessary for this call to have occurred, since every term in the data base is tried with every pattern. Note that we are relying on the fact that the bindings returned by match-w-equality will correspond to its argument t if t matches the pattern with no substitutions.

In certain circumstances, the algorithm in Figure 2 causes the same demon to be applied more than once to the same term. This inefficiency is eliminated in Section 4.6.

The algorithm in Figure 2 is complete for flat patterns and a fixed equality relation. It is clear by the definition of a variant, that the matcher is guaranteed to find a matching variant, if possible. In the case of flat patterns, all matching variants are equivalent, so only one matching variant is required to satisfy the condition of the Completeness Theorem.

In the next two sections, we extend this algorithm to the case of changing equalities, but still considering only flat patterns. Following this, we treat the case of nested patterns.

4.2 Changing Equalities

When a new equality is asserted, two previously separate equality classes are joined. This in turn makes new variants possible. Any term with a

subterm in either of the two equality classes must be re-matched against the patterns of all demons (see Figure 3). Matches which failed before may now succeed.

For example, suppose there is a demon with the pattern $f(?x, 1)$. When this pattern is first matched against the term $f(a, b)$, with $a=0$ and no other equalities, the match fails. If the equality $b=1$ is asserted, however, the term $f(a, b)$ needs to be re-matched (because it has b as a subterm). This time the match succeeds, yielding the matching variant $f(a, 1)$. In BREAD, this re-matching process is made more efficient by indexing terms and patterns according the subterms that appear in them.

When an equality is retracted, some of the variants of a given term matching a given pattern may cease being equal to the given term. This suggests that re-matching is also required when equalities are retracted.

For example, consider the set of variants of $f(a, b)$ matching the pattern $f(0, ?x)$, with $a=0$ and $b=1$:

$$\{f(0, b), f(0, 1)\}.$$

Since all of these equivalent, from the standpoint of the Completeness Theorem, it doesn't matter which is created for the demon to be applied to.

Suppose that the variant $f(0, 1)$ is chosen, and the demon from Section 1.1 is applied to it. The demon asserts

$$f(0, 1) \geq 0,$$

from which the system can deduce by equality reasoning that

$$f(a, b) \geq 0.$$

```
1  function join-equality-classes(C1, C2)
       [[ Join equality classes C1 and C2. ]]
2      for each term t in data base
3          if a subterm of t is in C1 ∪ C2
4              then for each demon ⟨ p , d ⟩
5                       try-w-equality(t, p, d)
```

Figure 3. Procedure to be executed whenever two equality classes are joined.

Now suppose that $b=1$ is retracted, but $a=0$ remains. The system is now incomplete. In order to regain completeness, the pattern $f(0,?x)$ needs to be re-matched against $f(a,b)$, in order to create the variant $f(0,b)$ and apply the demon to it. Note, however, that if $f(0,b)$ were chosen in the first place, there would be no need for re-matching.

The basic idea why it was a bad idea to choose the variant $f(0,1)$ above is because this variant uses more substitutions than necessary to match the pattern (e.g., substituting 1 for b was unnecessary). These extra substitutions make the deductive chain from the variant to the original term susceptible to be being broken when the irrelevant equalities (e.g., $b=1$) are retracted. We formalize this notion in the next section, and show that the algorithm in Figure 2 always chooses a matching variant (e.g., $f(0,b)$), such that completeness is guaranteed without re-matching on retraction of equalities.

4.3 Closest Matching Variants

The matching variant chosen by the algorithm in Figure 2 is as "close" as possible to the given term, in sense of depending on a minimum number of equalities. We call this variant a *closest* matching term, defined as follows.

> **Definition.** For a given term and flat pattern, a *closest matching term* is a matching term in which, for each variable in the pattern, the corresponding subterm in the matching term is identical to (at least) one of the corresponding subterms in the given term.

Note that the definition of closest matching term does not involve the equality relation. Also, there is not always a unique closest matching term. For example, both $f(a,a)$ and $f(b,b)$ are closest matching terms for the term $f(a,b)$ and the pattern $f(?x,?x)$. (Section 4.6 describes an improvement to the matching algorithm which uniquely chooses a closest matching variant.) For a term and pattern that match without any substitutions, the closest matching term is (uniquely) the term itself.

The important property of closest matching terms is given by the following theorem.

Theorem 2 (Closest Matching Term) *For a given term and flat pattern, if a closest matching term is not a variant, then there are no matching variants.*

The proof of this theorem is more easily understood using the contrapositive form: If there is any matching variant, then every closest matching term is a variant.

Let v be a variant of the term t matching the flat pattern p. Let c be a closest term to t matching p. Let $?x$ be a variable in p. Since v matches p, it must have the identical subterm v_x in each position corresponding to $?x$. Since c matches p, it must have the identical subterm, c_x, in each position corresponding to $?x$. Furthermore, since c is a closest matching term to t, c_x must be one of the subterms of t corresponding to $?x$. Since v is a variant of t, v_x must be equal to each subterm of t corresponding to $?x$; in particular, $v_x = c_x$. Since v and c both match p, they can differ only in the variable positions of p. We have shown that, for an arbitrary variable $?x$, the corresponding subterms v_x and c_x are equal; therefore c is a variant of v. Since v is a variant of t, c is also a variant of t.

This theorem guarantees that no re-matching needs to be done when an equality is retracted. Patterns and terms that did not match before certainly will not match after the retraction. For patterns and terms that did match before the retraction, the body of the demon was applied to a closest matching term. If that term is no longer a variant after retraction, this theorem guarantees that there are no matching variants, so there is no point in trying to re-match.

The use of the closest matching variant can lead to the creation of more than the minimum number of matching variants required to guarantee completeness. It is essentially a trade-off between creating extra variants when a term is added to the data base versus extra matching when an equality is retracted. This trade-off is discussed further in Section 5.1.

4.4 Flattening Nested Patterns

The match-w-equality algorithm yields at most a single variant. Since we have already seen examples in Section 3.2 involving nested patterns, wherein more than one variant is needed to guarantee completeness, this algorithm is clearly incomplete for the general case of nested patterns.

One might consider trying to directly generalize match-w-equality further to make it complete for nested patterns. This would entail matching against all members of equality classes at intermediate levels in a pattern. For example, in Section 3.2, when matching the pattern $P(h(?x))$ against the term $P(c)$, one would match the subpattern $h(?x)$ against all terms equal to c (not

only variants). In this case the equality class of c contains the terms $h(a)$ and $h(b)$.

In the presence of changing equalities, however, this approach is not satisfactory, since it does not keep track of intermediate equalities (equalities corresponding to intermediate levels in patterns). For example, suppose that $h(a) = h(b)$ is asserted only after the matching of $P(h(?x))$ against $P(c)$ had been performed. The variant $P(h(b))$ will not be created unless the matching is re-done. This would require all terms and patterns to be re-matched whenever any equalities are added, making the cost of incremental assertion prohibitive.

In this section, we show how to reduce the case of nested patterns to the case of flat patterns (for which we have a complete solution) by automatically defining intermediate demons. The basic idea of the reduction is to incrementally process nested patterns "from the inside out". When an attempt is made to add a demon with a nested pattern to the system, instead of adding it directly, we look inside the pattern to find a flat subpattern at some level of nesting. An intermediate demon is then added with that (flat) pattern and a body that "remembers" the pattern and body of the original demon. The rationale for this process is that, if there is no variant of a term in the data base matching the flat subpattern, then there certainly can't be a variant matching the whole pattern.

When the intermediate demon is invoked, it uses its substitutions to make a new pattern with fewer variables. If this pattern is still nested, a further intermediate demon will be added for a flat subpattern, and so on. This process stops when there are no nested patterns left, at which point the original body gets executed.

The algorithm which performs the incremental flattening process sketched above is shown in Figure 4. We will first explore its behavior with some examples, and then give a proof of its completeness in general.[8]

We illustrate the algorithm with the example from Section 3.2. The definition of the demon

$$\textbf{Rule } P(h(?x)) \implies P(h(?x)) \rightarrow Q(?x)$$

results in the call

[8]Since match-w-equality will now only be called on flat patterns, the recursion in the definition of match-w-equality can be eliminated simply by "unrolling" the recursive call one level and then removing the recursive case.

```
1  function flatten(p, d)
       [[ Add demon with possibly nested pattern p and body d. ]]
2      if p is nested
3        then choose a subpattern p_i of p
4             flatten(p_i, λtB . flatten(substitute(p, B), d))
5        else add-demon-w-equality(p, d)     [[ flat case ]]
```

Figure 4. The algorithm for adding a new demon with a pattern that may include subpatterns. Note that this is the tail-recursive definition of an iterative algorithm.

$$\text{flatten}(P(h(?x)), \mathit{body}),$$

which finds the flat subpattern $h(?x)$ and adds the intermediate demon

$$\langle\ h(?x)\ ,\ \lambda tB\,.\,\text{flatten}(\text{substitute}(P(h(?x)), B), \mathit{body})\ \rangle.$$

(Note that we are assuming here that both the matching term t and the corresponding bindings B are passed to the body of a demon.)

Given the term $P(c)$ and the equality relation[9]

$$\begin{aligned} c &= h(a), \\ c &= h(b), \end{aligned}$$

the pattern of the intermediate demon matches against the term $h(a)$ returning the binding $\{?x \leftarrow a\}$. The body of the demon substitutes this binding in the pattern $P(h(?x))$ and calls

$$\text{flatten}(P(h(a)), \mathit{body}).$$

This call to flatten goes directly to the flat case, adding the demon

$$\langle\ P(h(a))\ ,\ \mathit{body}\ \rangle.$$

[9]It does not matter whether this term and equalities were present at the time the demon was added to the system, or whether they were added afterwards, since the system has already been shown to be complete for demons with flat patterns, regardless of the order of events.

which, when matched against the term $P(c)$, causes the original *body* to be applied to the variant $P(h(a))$, asserting

$$P(h(a)) \rightarrow Q(a).$$

An analogous process starting with $h(b)$ causes the *body* to be applied to the variant $P(h(b))$. As discussed in Section 3.2, these are the two matching variants of $P(c)$ required for completeness.

Note that even if the equalities

$$\begin{aligned} a &= d \\ b &= e \end{aligned}$$

are supported (or are asserted later), no additional variants are created, because $h(d)$ and $h(e)$ are equivalent to $h(a)$ and $h(b)$, respectively, with respect to the pattern $h(?x)$.

Additional example traces of the flattening of nested patterns are given in Figures 5 and 6. Figure 5 illustrates the flattening of a complex pattern with multiple variables and multiple occurrences of variables. Figure 6 illustrates that the final intermediate demon does not always have a degenerate pattern with no variables—it only must be flat.

The completeness of the flattening algorithm is established by the following two theorems.

Theorem 3 *For a given term and demon, the flattening algorithm applies the body of the demon to at least one term from each equivalence class of matching terms.*

Theorem 4 *For a given term and (possibly nested) pattern, if all of the variants created by the flattening algorithm in any equivalence class stop being variants, then there are no matching variants in that class.*

The proofs of both theorems are based on the following observation. Let v be some variant of t which matches p. Let $?x$ be some variable in p, and let v_x be the corresponding subterm of v. Let p' be the first flat pattern containing $?x$ which is added by flatten, and v' be the subterm of v corresponding to p'. If v' is present in the data base, it will match p' and v_x will be substituted for $?x$ in further patterns generated in the process of obtaining a variant. If v' is *not* present in the data base, it must be a variant of some term s which is

Rule $g(f(?x,?y),f(?x,?z)) \Longrightarrow g(f(?x,?y),f(?x,?z)) = f(?x,g(?y,?z))$

Initial data base:

$$
\begin{array}{rcl}
 & g(r,s) & \\
r & = & f(a,b) \\
s & = & f(d,c) \\
a & = & d
\end{array}
$$

- flatten($g(f(?x,?y),f(?x,?z)), body$)
  - ▷ adds $\langle\ f(?x,?y)\ ,\ body'\ \rangle$
    where $body'$ is λtB . flatten(substitute($g(f(?x,?y),f(?x,?z)),B),body$)

- $f(a,b)$ matches $f(?x,?y)$ with bindings $\{?x \leftarrow a,\ ?y \leftarrow b\}$
  - ▷ $body'$ applied to $f(a,b)$
    - ▷ flatten($g(f(a,b),f(a,?z)), body$)
      - ▷ adds $\langle\ f(a,?z)\ ,\ body''\ \rangle$
        where $body''$ is λtB . flatten(substitute($g(f(a,b),f(a,?z)),B),body$)

- $f(d,c)$ matches $f(a,?z)$ with bindings $\{?z \leftarrow c\}$ yielding variant $f(a,c)$
  - ▷ $body''$ applied to $f(a,c)$
    - ▷ flatten($g(f(a,b),f(a,c)), body$)
      - ▷ adds $\langle\ g(f(a,b),f(a,c))\ ,\ body\ \rangle$

- $g(r,s)$ matches $g(f(a,b),f(a,c))$ yielding $g(f(a,b),f(a,c))$
  - ▷ $body$ applied to $g(f(a,b),f(a,c))$
    - ▷ asserts $g(f(a,b),f(a,c)) = f(a,g(b,c))$

Figure 5. A trace of the process by which a demon implementing the distributive law for f over g is invoked in an example situation.

$$\langle\ f(h(?x),?y)\ ,\ body\ \rangle$$

Initial data base:

$$f(c,d)$$

$$c = h(a)$$

- flatten($f(h(?x),?y), body$)
  - ▷ adds $\langle\ h(?x)\ ,\ body'\ \rangle$
    where $body'$ is λtB . flatten(substitute($f(h(?x),?y), B$), $body$)

- $h(a)$ matches $h(?x)$ with bindings $\{?x \leftarrow a\}$
  - ▷ $body'$ applied to $h(a)$
    - ▷ flatten($f(h(a),?y), body$)
      - ▷ adds $\langle\ f(h(a),?y)\ ,\ body\ \rangle$

- $f(c,d)$ matches $f(h(a),?y)$ with bindings $\{?y \leftarrow d\}$ yielding $f(h(a),d)$
  - ▷ $body$ applied to $f(h(a),d)$

Figure 6. An example trace of the execution of a demon with a nested pattern, in which the final intermediate demon does not have a degenerate pattern.

in the data base, since otherwise v could not be a variant of t. When match-w-equality is called on p' and s it must succeed, since there is a matching variant—v'. The bindings produced will have a subterm s_x of s for $?x$; since p' is flat, it must be the case that $s_x = v_x$ holds (this part is similar to the proof of the Closest Matching Term Theorem for the flat case). In both cases, the process must finally yield a variant of t, since one such a variant is known to exist.

We have shown that, for an arbitrary variable, a variant will be produced which has either the subterm of v corresponding to that variable or another term which is equal to it. Therefore that variant differs from v only in the positions occupied by variables in p, and thus is equivalent to v. Since v was an arbitrary variant, this shows that the algorithm will yield at least one variant from each equivalence class.

Now consider the contrapositive of Theorem 4. Suppose after some changes to the equality relation, there is some variant v of t which matches p. From the argument above it follows that some equivalent variant of v has been added by the cascading algorithm. This variant must still be equal to v and therefore also to t.

A final point to note about the flattening algorithm is that it can, in the worst case, lead to the creation of an exponential number of variants. Sometimes, all of these terms are necessary to guarantee completeness.

In the case of flat patterns, each match between a pattern and a term can, in the worst case, lead to the creation of a single new variant. Therefore, the number of variants created by the algorithm will not exceed the product of the number of patterns and the number of terms in the data base.

Consider, however, the nested pattern $p(h(?x_1), \ldots, h(?x_n))$ and the term $p(c, \ldots, c)$, with the equalities $c = h(a)$ and $c = h(b)$. The flattening algorithm will substitute both $h(a)$ and $h(b)$ for each occurrence of c in the term. If $h(a)$ and $h(b)$ are not variants, then the exponential number of terms thus created are in fact all necessary for completeness. However, if $h(a)$ and $h(b)$ are variants (i.e., because $a = b$), all the terms created would be equivalent and any one would suffice. The algorithm is thus much less than optimal in this (relatively pathological) case.

4.5 Termination

Any pattern-directed invocation system (even without equality) has the problem that it is possible to write demons that cause the system to go into an

Rule $h(h(?x)) \Longrightarrow h(h(?x)) = ?x$

Initial data base:

$$h(h(a))$$

- flatten$(h(h(?x)), body)$
  - ▷ adds $\langle\ h(?x)\ ,\ body'\ \rangle$
    where $body'$ is $\lambda tB\,.\,$flatten(substitute$(h(h(?x)), B), body)$

- $h(a)$ matches $h(?x)$ with bindings $\{?x \leftarrow a\}$
  - ▷ $body'$ applied to $h(a)$
    - ▷ flatten$(h(h(a)), body)$
      - ▷ adds $\langle\ h(h(a))\ ,\ body\ \rangle$

- $h(h(a))$ matches $h(h(a))$
  - ▷ $body$ applied to $h(h(a))$
    - ▷ asserts $h(h(a)) = a$

- $h(h(a))$ matches $h(?x)$ with bindings $\{?x \leftarrow h(a)\}$
  - ▷ $body'$ applied to $h(h(a))$
    - ▷ flatten$(h(h(h(a))), body)$
      - ▷ adds $\langle\ h(h(h(a)))\ ,\ body\ \rangle$

- $h(a)$ matches $h(h(h(a)))$ [since $a = h(h(a))$] yielding $h(h(h(a)))$
  - ▷ $body$ applied to $h(h(h(a)))$
    - ▷ asserts $h(h(h(a))) = h(a)$

Figure 7. A trace of the process by which a demon implementing the involutive law for h is invoked in an example situation. Note that this example involves the circular equality $h(h(a)) = a$.

- $h(h(h(a)))$ matches $h(?x)$ with bindings $\{?x \leftarrow h(h(a))\}$
  - ▷ *body'* runs on $h(h(h(a)))$
    - ▷ add-demon($h(h(h(h(a)))), body$)
      - ▷ installs $\langle\ h(h(h(h(a))))\ ,\ body\ \rangle$

- $h(h(a))$ matches $h(h(h(h(a))))$ [since $h(a) = h(h(h(a)))$] yielding $h(h(h(h(a))))$
  - ▷ *body* runs on $h(h(h(h(a))))$
    - ▷ asserts $h(h(h(h(a)))) = h(h(a))$

- ...

Figure 8. If matching variants created in try-w-equalities were added to the data base, then the example trace in Figure 7 would continue without terminating, as shown above.

infinite loop. For example, when the term $h(a)$ is added to the data base, the following apparently straightforward demon implementing the involutive law will create the terms $h(h(a))$, $h(h(h(a)))$, $h(h(h(h(a))))$, and so on without end:[10]

$$\textbf{Rule}\ h(?x) \Longrightarrow h(h(?x)) = ?x.$$

The problem with this demon is that it creates a new term in its body that matches its own pattern. Sometimes several demons can conspire to the get same effect, making the problem much less obvious. It is the user's responsibility to refrain from writing these kinds of demons.

Non-terminating demons can often be rewritten so that they terminate. For example, Figure 7 shows the version of the involutive law used in FRAPPE, in which a larger pattern is used to avoid the termination problem.

The example in Figure 7 also involves circular equalities and illustrates why it is important that matching variants (created in line 21 of Figure 2)

[10] This kind of non-terminating behavior is often called "counting", because it reminds us of the Peano axiomitization of the natural numbers.

are not added to the data base. As can be seen in Figure 8, if the matching variant $h(h(h(a)))$ created at the end of Figure 7, were added to the data base, it would successfully match against the pattern $h(?x)$, eventually causing the matching variant $h(h(h(h(a))))$ to be created, and so on ad infinitum.

Despite the fact that it is always possible to write non-terminating demons, we do want to make sure that our extensions to the standard algorithm do not introduce any additional non-terminating behavior. One way of saying this more precisely is to prove that if all demons defined by the user have empty bodies, then the system will always terminate.

For flat patterns, termination is obvious. There are a finite number of demons and a finite number of terms in the data base. The algorithm can, at worst, create one matching variant for each pair of term and demon. Since the bodies of the demons are empty, and add-term-w-equality is not called on the matching variant, the system stops when all of these variants have been created.

For nested patterns, we have to worry about the intermediate demons created by the flattening process. All of these demons have flat patterns, but their bodies are not empty. Non-terminating behavior could result if the execution of the bodies created new demons ad infinitum. We can see this will not be the case, however, by observing that if there are variables in the pattern p of an intermediate demon, then the body of the demon is of the form

$$\lambda tB\,.\,\text{flatten}(\text{substitute}(p, B), \mathit{body}),$$

where *body* is the original empty body. When a demon of this form is executed, the substitution of bindings B in the pattern p must reduce the number of variables in the pattern. Eventually, a sequence of such calls must eliminate all of the variables in any pattern. When that happens, the call to flatten will go directly to the flat case and install a demon whose pattern has no variables and whose body is the original empty body, at which point the process terminates.

4.6 Improvements to the Algorithm

This section describes two extensions to the algorithm in Figure 2, which improve its performance. First, under certain circumstances, the same demon may be applied more than once to the same term. This problem is remedied by the addition of some indexing information on each term. Second, the

```
1  function try-w-equality(t, p, d)
       [[ Match term t against pattern p with equalities
        and apply body d to t or matching variant, if successful. ]]
2      B ← match-w-equality(p, t, ∅)
3      if match succeeded
4          then v ← the-term(substitute(p, B))
5               if (v is not in data base and d is not in demon list of v)
6                  or (v ≡ t and not joining equality classes)
7                   then apply d to v [and B]
8                        if v ≢ t then add d to demon list of v
```

Figure 9. An improved version of try-w-equality (see Figure 2), in which no demon is applied more than once to the same term. A list has been introduced in lines 5 and 8 to keep track of the demons applied to a given term while it is not in the data base.

non-uniqueness of closest matching variants causes the algorithm to create unnecessary variants under certain circumstances. This problem is alleviated by the addition of an ordering on terms.

The circumstances under which the same demon $\langle\ p\ ,\ d\ \rangle$ is applied to the same term v more than once are as follows. Suppose that v is the matching variant created by try-w-equality(t, p, d), where $t \not\equiv v$. Try-w-equality applies d to v, but does not add v to the data base. It is possible for the same term v to be created by a different call, try-w-equality(t', p, d), where $t \not\equiv t' \not\equiv v$.[11] In this situation, d will be applied again to v. Alternatively, suppose that v is now explicitly added to the data base, i.e., add-term-w-equality(v) calls try-w-equality(v, p, d). Again, d will be applied to v.

The problem here is that we don't know what demons have been applied to a term while it is in the "limbo" state (i.e., created but not in the data base). This problem is straightforwardly remedied by keeping a list of the demons applied to a given term while it is in this state. This list can be cleared and the memory regained in add-term-w-equality, when the term is added to the data base.

Another case where the same demon may be applied to the same term

[11] For example, let p be $h(?x)$, v be $h(a)$, t be $j(a)$, and t' be $k(a)$, with $h = j = k$.

more than once is when joining two equality classes. In that case, all demons have been applied to all matching terms in the data base, and the reason for re-matching is only to create new variants if appropriate. In that case there is therefore no need to apply the demon to the term t even if it matches the demon's pattern.

The improved version of try-w-equality with these additions is shown in Figure 9. Note that the test $v \not\equiv t$ in line 8 checks for when try-w-equality is called from add-term-w-equality and t is being added to the data base, in which case there is no point in updating the demon list of t.

A second problem with the algorithm in Figure 2 is it can create more matching variants than necessary for patterns with multiple occurences of a given variable. For example, consider matching the term $f(a, b)$ against the pattern $f(?x, ?x)$ with $a = b$. Both $f(a, a)$ and $f(b, b)$ are closest matching variants of $f(a, b)$ with respect to this pattern. Due to the order of matching, the algorithm in Figure 2 arbitrarily chooses to create $f(a, a)$. Now suppose that the term $f(b, a)$ is added to the data base. This term has the same two closest matching variants with respect to $f(?x, ?x)$, but in this case the order of matching in the algorithm chooses to create $f(b, b)$. Since $f(a, a)$ and $f(b, b)$ are equivalent with respect to the pattern $f(?x, ?x)$, and both are variants of both $f(a, b)$ and $f(b, a)$, only one is required to guarantee completeness.

This problem can be alleviated, with a small increase in the complexity of the matching algorithm, by introducing an arbitrary (e.g., lexicographic or timestamp) order on terms. When the matcher has a choice of bindings for a variable, it always chooses the smallest term in the order (see Figure 10). Thus in the example above, if lexicographic order is used, $f(a, a)$ will be chosen as the closest matching variant for both $f(a, b)$ and $f(b, a)$. Note that as long as closest matching variants are used, it will sometimes be necessary to create more than one variant from each equivalence class. For example, if a, b, c and d are all equal, there is no one term which is closest to both $f(a, b)$ and $f(c, d)$ with respect to the pattern $f(?x, ?x)$. See however Section 5.1, where an alternative to using closest matching variants is discussed.

```
1  function match-w-equality(p, t, B)
     [[ Match pattern p to term t with bindings B with equalities.
       Complete for flat patterns only. ]]
2    case
3      p is a variable:
4        if p is bound to s in B
5          then if s = t
6                 then                          [[ same value ]]
7                   if t ≺ s then replace binding of p in B by t
8                   return B
9                 else fail                     [[ different value ]]
10         else return B ∪ {p ← t}               [[ new binding ]]
11     ...
```

Figure 10. An improved version of the matching algorithm from Figure 2. Line 7 has been added to guarantee that the matcher chooses the same matching variant for variant inputs. "$t \prec s$" means t is less than s in some order.

5 Conclusion

5.1 Alternate Approaches

The main goal of the algorithm developed in this paper has been to guarantee the completeness of pattern-directed invocation without creating the potentially infinite set of all possible variants. As part of the development, however, a number of decisions have been made trading off between time (matching and demon execution) and space (term creation). These decisions have been based on the relative cost of various computations in our implementation environment, and our expectations of use, such as the relative frequency of additions to the data base versus retractions. In this section, we briefly sketch two alternate decisions that could reasonably be made.

In Section 4.1, it was suggested that, when match-w-equality succeeds, rather than sometimes creating a matching variant, one might always apply the given demon to the given term. For example, the term $f(a,b)$, with $a=0$, successfully matches against the demon

$$\langle\ f(0,?x)\ ,\ \lambda t\,.\,\mathrm{assert}(t \geq 0)\ \rangle,$$

returning the bindings $\{?x \leftarrow b\}$. Rather than creating the new term $f(0,b)$, one might consider applying the demon body to $f(a,b)$ directly, causing it to assert $f(a,b) \geq 0$, which is the desired conclusion in this case.

The problem with this approach is that the proper dependencies are not installed. The conclusion $f(a,b)$ needs to be retracted when the equality $a=0$ is retracted. In order to make this approach work correctly, we would need to supply the body of each demon with with the set of equalities required to make the match succeed.[12] Any conclusions made by the body would then have to depend on these equalities.

The attraction of this approach is that it can reduce the number of terms created. On the other hand, it can increase the number of demons applied. For example, suppose that in the situation above it is also the case that $c=0$, and the term $f(c,b)$ is in the data base. The algorithm developed in this paper will create the term $f(0,b)$, which is the closest matching variant to both $f(a,b)$ and $f(c,b)$, and apply the demon to it, asserting $f(0,b) \geq 0$. Both desired conclusions, namely $f(a,b) \geq 0$ and $f(c,b) \geq 0$ follow by substitution

[12] In the equality system of BREAD, it is very cheap to test if two terms are in the same equality class. It is much more expensive, however, to compute the supporting set of equalities.

of equals. The alternate approach suggested here will apply the demon twice, once to $f(a,b)$ and once to $f(c,b)$ to get the same conclusions.

Another way to reduce the number of terms created, mentioned in Section 4.3, is to give up using the closest matching variant. For example, consider matching the term in the data base $f(a,b)$ against the pattern $f(0,?x)$ with $a=0$. The closest matching variant in this case is $f(0,b)$—if it doesn't already exist, the algorithm developed above will create it. However, suppose the term $f(0,c)$ already exists (in the data base or in "limbo"), with $b=c$. This term is also a variant of $f(a,b)$ matching the pattern. Since all variants are equivalent in the flat case, completess is satisfied as long as the demon is applied to $f(0,c)$. The term $f(0,b)$ does not need to be created.

The problem with this approach, of course, is that when equality classes are split (e.g., when $b=c$ is retracted), re-matching needs to be done, similar to when equality classes are merged. At that time, another existing matching variant may be found, or new term may need to be created. Whether this trade-off is advantageous depends on the relative frequency of additions versus retraction.

Finally, a natural question to ask is whether the algorithm developed here can be generalized to unification, i.e., allowing terms in the data to contain variables (under the usual convention that free variables are universally quantified). Although the matcher could be straighforwardly generalized to do unification, it turns out not to be of much use.

To illustrate, suppose that the data base contains the term $f(?u,?v)$. This term would unify with the pattern of the following demon:

$$\textbf{Rule } f(?x,?x) \implies f(?x,?x)=?x.$$

However, creating the unifying term $f(?u,?u)$ and applying the demon to it would cause $f(?u,?u)=?u$ to be asserted, which doesn't add any new information about the original term $f(?u,?v)$. The conclusion to be made here is that, in this setting, logical variables and pattern variables are different entities.

While on the topic of unification, it is also worth pointing out related research on so-called "generalized unification" [9]. The problem studied in this work is unification/matching in the presence of implicit equalities due to some fixed theory, such as commutativity or associativity. In contrast, our work is concerned with matching in the context of an explicit data base of arbitrary, changing equalities.

5.2 Summary

Pattern-directed invocation in the presence of arbitrary equalities raises the completeness problem addressed in this paper. If the equalities are fixed and patterns are flat, the changes that need to be made to the standard pattern-directed invocation algorithm are relatively straightforward. Changing equalities and nested patterns introduce most of the complexity of the algorithm developed here.

The algorithm described here has been implemented in a working system called BREAD. This system has proved to be a useful foundation upon which we have built a frame system [2] and a system for reasoning about programs [7].

References

[1] J. Doyle. A truth maintenance system. *Artificial Intelligence*, 12:231–272, 1979.

[2] Y. A. Feldman and C. Rich. FRAPPE: Frames in a propositional environment. In preparation, 1988.

[3] D. A. McAllester. An outlook on truth maintenance. Memo 551, MIT Artificial Intelligence Lab., August 1980.

[4] D. A. McAllester. The use of equality in deduction and knowledge representation. Technical Report 550, MIT Artificial Intelligence Lab., January 1980. Master's thesis.

[5] D. A. McAllester. Reasoning utility package user's manual. Memo 667, MIT Artificial Intelligence Lab., April 1982.

[6] G. Nelson and D. C. Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM*, 27(2):356–364, April 1980.

[7] C. Rich. The layered architecture of a system fo reasoning about programs. In *Proc. 9th Int. Joint Conf. Artificial Intelligence*, pages 540–546, Los Angeles, CA, 1985.

[8] C. Rich and R. C. Waters. The programmer's apprentice project: A research overview. Memo 1004, MIT Artificial Intelligence Lab., November

1987. Submitted to IEEE Software/IEEE Expert Special Issue on the Interactions between Expert Systems and Software Engineering.

[9] J. H. Siekmann. Universal unification. In R.E. Shostak, editor, *Proc. 7th Int. Conf. on Automated Deduction*, pages 1–42. Springer Verlag, New York, 1984. Lecture Notes in Computer Science Series, Vol. 170.

END

DATE

FILMED

DTIC

10-88